Semantic Macros
Attribute Grammar Combinators

Marcos Viera

S. Doaitse Swierstra

Technical Report UU-CS-2011-028
Sept 2011

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Semantic Macros
Attribute Grammar Combinators

Marcos Viera[1] and Doaitse Swierstra[2]

[1] Instituto de Computación, Universidad de la República
Montevideo, Uruguay
mviera@fing.edu.uy
[2] Department of Computer Science, Utrecht University
Utrecht, The Netherlands
doaitse@cs.uu.nl

Abstract. Having extensible languages is appealing but raises the question of how to construct extensible compilers and how to compose compilers out of a collection of pre-compiled components.
Being able to deal with attribute grammar fragments as first-class values makes it possible to describe semantics in a compositional way; this leads naturally to a plug-in architecture, in which a core compiler can be constructed as a (collection of) pre-compiled component(s), and to which extra components can safely be added at will as need arises.
We present an Emddded Domain Specific language (EDSL), in the form of a Haskell combinator library, which makes the above possible in a *typeful way*; both the check for the well-definedness of the constructed attribute grammar and the well-definedness of attribute computations is taken care of by the Haskell type checker.
With our combinators it is easy to describe semantics in terms of already existing semantics, just as syntax macros extend language syntax. We also show how existing semantics can be redefined, thus adapting some aspects from the behavior defined by the macros.

1 Introduction

Since the introduction of the very first programming languages, and the invention of grammatical formalisms for describing them, people investigate how an initial language definition can be extended by someone else than the original language designer by including separate, pre-compiled language-definition fragments.

The simplest approach starts from the *compiler text* corresponding to the base language. Just before the compiler is compiled, several extra ingredients can be added textually. In this way we get great flexibility and there is virtually no limit to the things we may add. The Utrecht Haskell Compiler [5] has shown the effectiveness of this approach by composing attribute grammar fragments textually into a complete compiler description. This approach however is not very practical when defining relatively small language extensions; we do not want every individual user to generate a completely new compiler for each small extension. Another problematic aspect of this approach is that, by making the complete text of the compiler available for modification or extension, we may also loose important guarantees provided by e.g. the type system; we definitely do not want everyone to mess around with the delicate internals of a compiler for a complex language.

So the question arises how we can reach the same effect but without opening up the whole compiler source. The most commonly found approach is to introduce so-called *syntax macros* [9], which enable the programmer add *syntactic sugar* to a language by defining new notation *in terms of already existing syntax*.

In this paper we will focus in how to provide such mechanisms *at the semantic level*. As a running example we take a minimal expression language described by:

$$\textbf{data}\ \mathit{Root} = \mathit{Root}\ \{\mathit{expr} :: \mathit{Expr}\}$$
$$\textbf{data}\ \mathit{Expr} = \mathit{Cst}\ \ \{\mathit{cv}\ \ \ :: \mathit{Int}\}$$

$$
\begin{array}{l}
\mid \ \mathit{Var} \ \ \{\mathit{vnm} :: \mathit{String}\} \\
\mid \ \mathit{Mul} \ \ \{\mathit{me1} :: \mathit{Expr}, \ \ \mathit{me2} :: \mathit{Expr}\} \\
\mid \ \mathit{Add} \ \ \{\mathit{ae1} \ :: \mathit{Expr}, \ \ \mathit{ae2} \ :: \mathit{Expr}\} \\
\mid \ \mathit{Let} \ \ \{\mathit{lnm} \ :: \mathit{String}, \mathit{val} \ \ :: \mathit{Expr}, \mathit{body} :: \mathit{Expr}\}
\end{array}
$$

Suppose we want to extend the language with one extra production for defining the square of a value. A syntax macro aware compiler might be willing to accept definitions of the form $\mathit{square}\ (\mathit{se} :: \mathit{Expr}) \Rightarrow \mathit{Mul}\ \mathit{se}\ \mathit{se}$, translating new syntax into the core abstract syntax.

Despite the fact that this approach may be very effective, such transformational programming [3] has severe shortcomings; as a consequence of mapping the new constructs onto existing constructs and performing any further processing such as type checking on this simpler, but often more detailed program representation, feedback from later stages is given in terms of invisible intermediate program representations. For example, if we do not change the pretty printing phase of the compiler *square* 2 might be printed as $2 * 2$. Hence the implementation details shine through, and the produced error messages can be confusing or even incomprehensible. Similar problems show up when defining embedded domain specific languages: the error messages are typically given in terms of the underlying presentation [6].

In a previous paper [13] we introduced a Haskell library of first-class attribute grammars, which can be used to implement a language semantics and its extensions in a safe way, i.e. by constructing a core compiler as a (collection of) pre-compiled component(s), and to which extra components can safely be added at will. In this paper we show how we can define the semantics of the right hand side in terms of existing semantics, in the form of *attribute grammar macros*.

We also show how, by using first class attribute grammars, the already defined semantics can be *redefined* at the places where it makes a difference, e.g. in pretty printing and generating error messages.

In Section 2 we describe our approach to first-class attribute grammars, and Section 3 we show how to define semantic macros and redefine attributes. We close by presenting our conclusions and future work. Figures 2, 4 and 5 contain the code describing the semantics of our example language, and code for the extension is given in figures 6 and 11.

2 AspectAG

In this section we describe *AspectAG*[13], a Haskell library[3] for defining first-class attribute grammars. The key technique uderlying our embedded approach lies in using the Hlist library [8] for typed heterogeneous collections (extensible polymorphic records) for representing collections of attributes, and expressing the AG well-formedness conditions by type-level predicates (i.e., type-class constraints), thus mimicking dependently typed programming techniques in Haskell [10] by using Haskell 98 extensions such as multi-parameter type classes and functional dependencies [7]; *type-level programming* uses types to represent type-level values, and class instances to represent type-level functions.

Heterogeneous lists are constructed using the functions $(.*.)$ and *hNil*, modeling the structure of a normal list both at the value and type level. An *extensible record* is an heterogeneous list of uniquely labeled fields marked with the type *Record*. A field $(l \ .=. \ v)$ relates a (first-class) label l with the value v. Extensible records can be constructed with the functions $(.*.)$ and *emptyRecord*; where $(.*.)$ is overloaded to not only extend the list both at type and value level, but also to impose by (type class) constraints that elements in a record are uniquely labeled. In order to keep our programs readable we will use the following syntactic sugar to denote lists and records in the rest of the paper:

- $\{\ v_1, ..., v_n\ \}$ for $(v_1\ .*.\ ...\ .*.\ v_n\ .*.\ \mathit{HNil})$
- $\{\{\ v_1, ..., v_n\ \}\}$ for $(v_1\ .*.\ ...\ .*.\ v_n\ .*.\ \mathit{emptyRecord})$

[3] http://hackage.haskell.org/package/AspectAG

Thus, if $label_1$ and $label_2$ are labels, the following is the definition of a record (myR) with the elements *True* and `"bla"`:

$$myR = \{\{\ label_1 \mathrel{.=.} True,\ label_2 \mathrel{.=.} \texttt{"bla"}\ \}\}$$

The operator (#) is used to retrieve the value part corresponding to a specific label from a record, statically enforcing that the record indeed has a field with this label. The expression ($myR\#label_2$) returns the string `"bla"`, while, given a label $label_3$, the expression ($myR\#label_3$) does not compile.

2.1 Rules

In this subsection we show how attributes and their defining rules are represented. An *attribution* is a finite mapping from attribute names to attribute values, represented by a *Record*, in which each field represents the name and value of an attribute.

When inspecting what happens at a production (a node of the abstract syntax tree) we see that information flows from the inherited attribute of the parent (ip) and the synthesized attributes of the children (sc) to the synthesized attributes (sp) of the parent and the inherited attributes of the children (ic). Henceforth the attributes ip and sp together are called *input family* while the attributes sp and ic are called *output family*, both represented by:

data *Fam children parent* = *Fam children parent*

A *Fam* contains a single attribution for the parent and a collection of attributions for the children. Hence the type *parent* will always be a *Record* with fields labeled by attribute names; thus the type *children* will be a *Record* with fields labeled by children names and attributions (*Record*s) as values.

In Figure 2 we start with the Template Haskell functions *addNont* and *addProd*, from *AspectAG*, which introduce the labels for the abstract syntax of our example. The calls to *addNont* generate the labels *nt_Root* and *nt_Expr* so we can refer to the respective non-terminals, and the empty types *Root* and *Expr*. The calls to *addProd* generate the labels of the children of the productions; the type of each label encodes both the name and the type of the child. For example, for the *Root* production a single label *ch_expr* will be generated, indicating at the type level that this child is of type *Expr*. Here we could have generated all this from the data type definition describing the abstract syntax.

Attribute computations are defined in terms of *rules*. As defined by [4], a rule is a mapping from an input family to an output family. In order to make rules composable we define a rule as a mapping from input attributes to a function which extends a family of output attributes with the new elements defined by this rule:

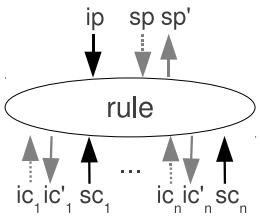

Fig. 1. Rule: black arrows represent input and grey arrows represent output; dotted grey arrows represent the already constructed output which can be used to compute further output elements (hence the direction of the arrow)

type *Rule sc ip ic sp ic′ sp′* = *Fam sc ip* → (*Fam ic sp* → *Fam ic′ sp′*)

Thus, the type *Rule* states that a rule takes as input the synthesized attributes of the children *sc* and the inherited attributes of the parent *ip* and returns a function from the output constructed thus far (inherited attributes of the children *ic* and synthesized attributes of the parent *sp*) to the extended output. Figure 1 shows a graphic representation of rules; each rule can be seen as a node of a graph with an underlying three-shaped structure, determining the flow of the attributes.

2.2 Rule Definition

The functions *syndefM* and *inhdefM* are used to define a single rule defining a synthesized or an inherited attribute respectively. Figure 2 lists all the rule definitions of the static semantics of our running example. It defines two aspects: pretty printing, realized by a synthesized attribute *spp*, which holds a pretty printed document of type *PP_Doc*, and expression evaluation, realized by two attributes: a synthesized *sval* of type *Int*, which holds the result of an expression, and an inherited *ienv* which holds the environment ($[(String, Int)]$) in which an expression is to be evaluated. In our naming convention a rule with name *attProd* defines the attribute *att* for the production *Prod*. The rule *sppLet* for the attribute *spp* of the production *Let* combines the strings `"let"`, *lnm* and `"="`, the pretty printed child *val*, the string `"in"` and the pretty printed child *body*, using the pretty printing combinator (`>#<`)for horizontal (beside) composition, from the *uulib*[4] library. The rule *ienvLet* specifies that the *ienv* value coming from the parent (*lhs* stands for "left-hand side") is copied to the *ienv* position of the child *val*; the *ienv* attribute of the *body* is this environment extended with a pair composed of the name (*lnm*) associated with the child and the value (the *sval* attribute) of the child. The rule *svalLet* defines the attribute *sval* to be the attribute *sval* of the *body*.

The functions *syndefM* and *inhdefM* are versions of *syndef* and *inhdef*, that use a *Reader* monad to make definitions look somewhat "prettier".

The function *syndef* adds the definition of a synthesized attribute. It takes a label *att* representing the name of the new attribute, a value *val* to be assigned to this attribute, and it builds a function which updates the output for the father as constructed thus far.

$$syndef\ att\ val\ (Fam\ ic\ sp) = Fam\ ic\ (att\ \texttt{.=.}\ val\ \texttt{.*.}\ sp)$$

$$syndefM\ att\ mval\ inp = syndef\ att\ (runReader\ mval\ inp)$$

The record *sp* which holds the synthesized attributes of the parent is extended with a field with name *att* and value *val*.
Let us take a look at how the rule definition *sppAdd* of the attribute *spp* for the production *Add* is defined using *syndef*:

$$sppAdd\ (Fam\ sc\ ip)$$
$$= syndef\ spp\ \$\ ((sc\ \#\ ch\_ae1)\ \#\ spp)\ \texttt{>\#<}\ \texttt{"+"}\ \texttt{>\#<}\ ((sc\ \#\ ch\_ae2)\ \#\ spp)$$

The children *ch_ae1* and *ch_ae2* are retrieved from the input family so we can subsequently retrieve the attribute *spp* from these attributions, and construct the computation of the synthesized attribute *spp*.

The function *inhdef* introduces a new inherited attribute for a collection of non-terminals at the same time.

$$inhdef\ ::\ Defs\ att\ nts\ vals\ ic\ ic'$$
$$\Rightarrow att \rightarrow nts \rightarrow vals \rightarrow (Fam\ ic\ sp \rightarrow Fam\ ic'\ sp)$$

It results in a function which updates the output constructed thus far and takes the following parameters: the attribute *att* which is being defined, the list *nts* of non-terminals with which this attribute is being associated, and a record *vals* labeled with child names and containing values,

[4] http://hackage.haskell.org/package/uulib

```
  -- Abstract Syntax
$ (addNont "Root")
$ (addNont "Expr")
$ (addProd "Root" [("expr", "Expr")])
$ (addProd "Cst"  [("cv",   "Int")])
$ (addProd "Var"  [("vnm", "String")])
$ (addProd "Mul"  [("me1", "Expr"), ("me2", "Expr")])
$ (addProd "Add"  [("ae1", "Expr"), ("ae2", "Expr")])
$ (addProd "Let"  [("lnm", "String"), ("val", "Expr"), ("body", "Expr")])
  -- Pretty Printing
$ (attLabels ["spp"])
sppRoot = syndefM spp $ liftM (#spp) (at ch_expr)
sppCst  = syndefM spp $ liftM pp     (at ch_cv)
sppVar  = syndefM spp $ liftM pp     (at ch_vnm)
sppMul  = syndefM spp $ do e1 <- at ch_me1
                           e2 <- at ch_me2
                           return $ e1 # spp >#< "*" >#< e2 # spp
sppAdd  = syndefM spp $ do e1 <- at ch_ae1
                           e2 <- at ch_ae2
                           return $ e1 # spp >#< "+" >#< e2 # spp
sppLet  = syndefM spp $ do lnm  <- at ch_lnm
                           val  <- at ch_val
                           body <- at ch_body
                           return $ "let" >#<
                                    lnm   >#< "=" >#< val # spp >#<
                                    "in"  >#< body # spp
  -- Evaluation
$ (attLabels ["ienv", "sval"])
  -- Environment
ienvRule = copy ienv { nt_Expr }
ienvRoot = inhdefM ienv { nt_Expr } $
                   do return {{ ch_expr .=. ([] :: [(String, Int)]) }}
ienvMul  = ienvRule
ienvAdd  = ienvRule
ienvLet  = inhdefM ienv { nt_Expr } $
                   do lnm  <- at ch_lnm
                      val  <- at ch_val
                      lhs  <- at lhs
                      return {{ ch_val  .=. lhs # ienv
                              , ch_body .=. (lnm, val # sval) : lhs # ienv }}
  -- Value
svalRule f = use sval { nt_Root, nt_Expr } f (0 :: Int)
svalRoot = syndefM sval $ liftM (#sval) (at ch_expr)
svalCst  = syndefM sval $ liftM id (at ch_cv)
svalVar  = syndefM sval $ do vnm <- at ch_vnm
                             lhs <- at lhs
                             return $ fromJust (lookup vnm (lhs # ienv))
svalMul  = svalRule (*)
svalAdd  = svalRule (+)
svalLet  = syndefM sval $ liftM (#sval) (at ch_body)
```

Fig. 2. AspectAG specification

describing how to compute the attribute being defined at each of the applicable child positions. The class *Defs* is a type-level function used to iterate over the record *vals* and to compute the new record of inherited attributes ic', extending the record *ic* with the inherited attributes defined thus far.

Thus, in our example, *ienvLet* defines the computation of the attribute *ienv* (environment) for the production *Let*. We give a definition for the attribute *ienv* for each child of which the semantic category is in the list {{ *nt_Expr* }}, and these are stored in an extensible record labeled by the names of the children.

Explicitly giving all rules soon becomes cumbersome, so handy shortcuts are available: *copy* rules and *use* rules. A copy rule copies an inherited attribute from a parent to all its children. The function *copy* takes the name of the attribute and an heterogeneous list of the non-terminals for which the attribute has to be defined, and generates copy rules for these. A copy rule is used for example in the definition of *ienvAdd*, in Figure 2, instead of writing the explicit code:

```
ienvAdd = inhdefM ienv { nt_Expr } $ do lhs ← at lhs
                                       return {{ ch_ae1 .=. lhs # ienv
                                               , ch_ae2 .=. lhs # ienv }}
```

A use rule introduces a synthesized attribute that collects information from some of the children. The function *use* takes the following arguments: the attribute being defined, the list of semantic categories for which the attribute is to be defined, a monoidal operator which combines the attribute values, and a unit value to be used in those cases where none of the children has such an attribute. Thus, the definition of *svalAdd* in Figure 2 is equivalent to:

```
svalAdd = syndefM sval $ do e1 ← at ch_ae1
                            e2 ← at ch_ae2
                            return $ (e1 # sval) + (e2 # sval)
```

2.3 Rules Composition

The composition of two rules is the composition of the two functions resulting from applying each of them to the input family:

```
ext :: Rule sc ip ic' sp' ic'' sp'' → Rule sc ip ic sp ic' sp'
    → Rule sc ip ic sp ic'' sp''
(rule1 `ext` rule2) input = rule1 input.rule2 input
```

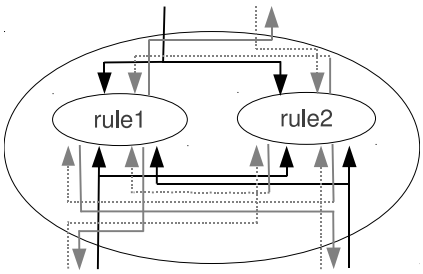

Fig. 3. Rules Composition: produces a new rule, represented by the external oval

Figure 3 represents a composition *rule1* `ext` *rule2*, of rules with two children. By solving the labyrinths of this figure, it can be seen how the inputs are shared and the outputs are combined by using the outputs of *rule2* as output constructed thus far of *rule1*. In Figure 4 we show for each production of the example how we combine its attributes using the function *ext*.

```
aspRoot = sppRoot `ext` svalRoot `ext` ienvRoot
aspCst  = sppCst  `ext` svalCst
aspVar  = sppVar  `ext` svalVar
aspMul  = sppMul  `ext` svalMul  `ext` ienvMul
aspAdd  = sppAdd  `ext` svalAdd  `ext` ienvAdd
aspLet  = sppLet  `ext` svalLet  `ext` ienvLet
```

Fig. 4. Aspects

2.4 Semantic Functions

The semantics we associate with an abstract syntax tree is a function which maps the inherited attributes of the root node to its synthesized attributes. So for each production we now construct a function that takes the semantics of its children and maps it to the semantics of the resulting tree rooted at the node where this production was applied. We will refer to such functions as *semantic functions*. The hard work is done by the function *knit*, that "ties the knot", combining the attribute computations with the semantics of the children trees (describing the flow of data from their inherited to their synthesized attributes) into the semantic function for the parent. In Figure 5 we show the definition of the semantic functions of the example, where the function *knit* is applied to the combined attributes for the production.

```
semRoot_Root sexpr = knit aspRoot {{ ch_expr .=. sexpr }}
semExpr_Cst  scv   = knit aspCst  {{ ch_cv .=. scv }}
semExpr_Var  svnm  = knit aspVar  {{ ch_vnm .=. svnm }}
semExpr_Mul  sme1 sme2 = knit aspMul {{ ch_me1 .=. sme1, ch_me2 .=. sme2 }}
semExpr_Add  sae1 sae2 = knit aspAdd {{ ch_ae1 .=. sae1, ch_ae2 .=. sae2 }}
semExpr_Let  slnm sval sbody = knit aspLet {{ ch_lnm .=. slnm, ch_val .=. sval
                                            ,  ch_body .=. sbody }}
```

Fig. 5. Semantic Functions

3 Attribute Grammar Combinators

Thus far we have described an EDSL that allows us to define the static semantics of a language. The goal of this paper is to show how we can define new productions by combining existing productions, while probably updating some of the aspects. We want to express the semantics of new productions *in terms of already existing semantics* and *by adapting parts of the semantics* resulting from such a composition.

3.1 Attribute Grammar Macros

In Figure 6 we extend the language of our example with some extra productions; one for defining the square of a value, one for defining the sum of the squares of two values, and one for doubling a value.

The square of a value is the multiplication of this value by itself. Thus, the semantics of multiplication can be used as a basis, by passing to it the semantics of the only child (*ch_se*) of the square production both as *ch_me1* and *ch_me2*. We do so in the definition of *aspSq* in Figure 6;

```
$ (addProd "Sq" [("se", "Expr")])
aspSq    = agMacro (aspMul  ,  ch_me1 ↪ ch_se
                            <.> ch_me2 ↪ ch_se)
$ (addProd "Pyth" [("pe1", "Expr"), ("pe2", "Expr")])
aspPyth = agMacro (aspAdd  ,  ch_ae1 ⟹ (aspSq, ch_se ↪ ch_pe1)
                            <.> ch_ae2 ⟹ (aspSq, ch_se ↪ ch_pe2))
$ (addProd "Double" [("de", "Expr")])
aspDouble = agMacro (aspMul  ,  ch_me1 ⟹ (aspCst, ch_cv ⇝ 2)
                              <.> ch_me2 ↪ ch_de)
```

Fig. 6. Language Extension

we declare an *attribute grammar macro* based on the attribute computations for the production *Mul*, defined in *aspMul*, with its children (*ch_me1* and *ch_me2*) mapped to the new child *ch_se*.

Attribute macros can map children to other macros, and so on. For example, in the definition of *aspPyth* (sum of the squares of *ch_pe1* and *ch_pe2*) the children are mapped to macros based on the semantics of square (*aspSq*).

When defining a macro based on the semantics of a production which has literal children, these children can be mapped to literals. In the definition of *aspDouble* the child *ch_me1* of the multiplication is mapped to a constant, which is mapped to the literal 2.

An attribute grammar macro is determined by a pair with the *base rule* ($rule_b$) of the macro and the mapping (*chMap*) between the children of this rule and their newly defined semantics, and returns a *macro rule*. As shown in Figure 7, *chMap* (rectangle) is an interface between the children of the base rule (inner oval) and the children of the macro rule (outer oval). The number of children of the macro rule (below *chMap* in the figure) does not need to be the same as the number of children of the base rule.

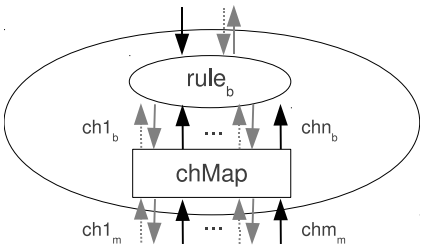

Fig. 7. AG Macro

The function *agMacro* constructs the macro rule; it performs the "knitting" of $rule_b$, by applying this rule to its input and the output produced thus far. These elements have to be obtained from the corresponding elements of the macro rule and the mapping *chMap*. To keep the code clear, we will use the subindex b for the elements of the base rule and m for the elements of the macro rule. Thus, the macro rule takes as input the family ($Fam\ sc_m\ ip_m$) and updates the output family constructed thus far ($Fam\ ic_m\ sp_m$) to a new output family ($Fam\ ic''_m\ sp'_m$):

$$
\begin{aligned}
&agMacro\ (rule_b, chMap)\ (Fam\ sc_m\ ip_m)\ (Fam\ ic_m\ sp_m) = \\
&\quad \textbf{let}\ ip_b = ip_m \\
&\qquad sp_b = sp_m \\
&\qquad (Fam\ ic'_b\ sp'_b) = rule_b \quad (Fam\ sc_b\ ip_b)\ (Fam\ ic_b\ sp_b)
\end{aligned}
$$

$$
\begin{aligned}
&(ic'_m, ic_b, sc_b) \;=\; chMap\ (sc_m, ic_m)\ (ic'_b, emptyRecord, emptyRecord)\\
&ic''_m \;=\; hRearrange\ (recordLabels\ ic_m)\ ic'_m\\
&sp'_m = sp'_b\\
\textbf{in}\ \ &(Fam\ ic''_m\ sp'_m)
\end{aligned}
$$

The inherited and synthesized attributes of the parent of the base rule (ip_b and sp_b) respectively correspond to ip_m and sp_m, the inherited and synthesized attributes of the parent of the macro rule. The inherited and synthesized attributes of the children of the base rule (ic_b and sc_b), as well as the updated inherited attributes of the children of the macro rule (ic'_m), are generated by the children mapping function *chMap*. The function *chMap* takes as input a pair (sc_m, ic_m) with the synthesized attributes and the inherited attributes constructed thus far of the children of the macro rule, an returns a function that updates a triple with the updated inherited attributes (ic'_m) of the children of the macro rule and the inherited (ic_b) and synthesized (sc_b) attributes of the children of the base rule. We define as the "initial" triple to update, a triple composed by the updated inherited attributes of the children of the base rule (ic'_b), which with some changes will be converted into ic'_m, and two empty records (to be extended to ic_b and sc_b). Notice that the attributes we pass to *chMap* are effectively the ones indicated by the incoming arrows in Figure 7.

The rearranging of ic'_m is just a technical detail due to the use if *HList*; by doing this we assure that the children in ic_m and ic'_m are in the same order, thus explaining to the type system that both represent the same production. The synthesized attributes of the parent of the macro rule (sp'_m) are just sp'_b, the synthesized attributes of the parent of the base rule.

A mapping function is similar to rules in the sense that they take an input and return a function that updates its "output", that in this case is the triple (ic'_m, ic_b, sc_b) instead of an output family. Thus, they can be combined in the same way rules are combined; the combinator (<.>), used in Figure 6, is exactly the same as the *ext* function but with different types:[5]

$$
\begin{aligned}
&(<.>)\ ::\ ((sc_m, ic_m) \to ((ic'1_m, ic1_b, sc1_b) \to (ic'2_m, ic2_b, sc2_b)))\\
&\quad \to ((sc_m, ic_m) \to ((ic'0_m, ic0_b, sc0_b) \to (ic'1_m, ic1_b, sc1_b)))\\
&\quad \to ((sc_m, ic_m) \to ((ic'0_m, ic0_b, sc0_b) \to (ic'2_m, ic2_b, sc2_b)))\\
&(chMap1\ <.>\ chMap2)\ inp = chMap1\ inp.chMap2\ inp
\end{aligned}
$$

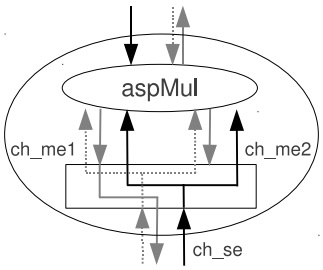

Fig. 8. aspSq

We use the combinator ($\hookrightarrow$) to map a child lch_b of the base rule to a child lch_m of the macro rule.

$$
\begin{aligned}
&lch_b \hookrightarrow lch_m = \lambda(sc_m, ic_m)\ (ic'0_m, ic0_b, sc0_b) \to\\
&\quad \textbf{let}\ ic'1_m = hRenameLabel\ lch_b\ lch_m\ (hDeleteAtLabel\ lch_m\ ic'0_m)\\
&\qquad ic1_b \;\; = lch_b\ .=.\ (ic_m\ \#\ lch_m)\ .*.\ ic0_b
\end{aligned}
$$

[5] To avoid confusion with rule combination, instead of using apostrophes to denote updates we use numeric suffixes

$$
\begin{aligned}
&sc1_b \;\; = lch_b \mathbin{.=.} (sc_m \mathbin{\#} lch_m) \mathbin{.*.} sc0_b \\
\textbf{in}\;\; &(ic'1_m, ic1_b, sc1_b)
\end{aligned}
$$

The updated inherited attributes for the child lch_m correspond to the updated inherited attributes of the child lch_b. Thus, the new ic'_m ($ic'1_m$) is the original one with the field lch_b renamed to lch_m. Since more than one child of the base rule can be mapped to a child of the macro rule, like in *aspSq* of Figure 6, we have to avoid duplicates in the record by deleting a possible previous occurrence of lch_m. This decision fixes the semantics of multiple occurrences of a child in a macro: the child will receive the inherited attributes of its left-most mapping. We represent this behavior in Figure 8 with the gray arrow, which corresponds to the inherited attributes of *ch_me2*, pointing nowhere outside the mapping. In the cases of the initial inherited attributes and the synthesized attributes, they have to be extended with a field corresponding to the child lch_b with the attributions for the child lch_m from the inherited and synthesized attributes, respectively, of the macro rule.

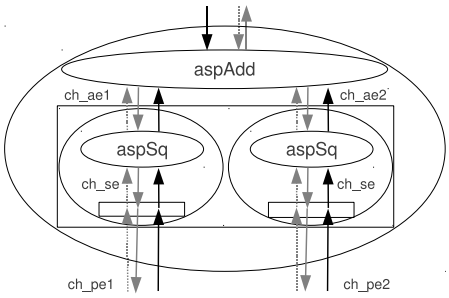

Fig. 9. aspPyth

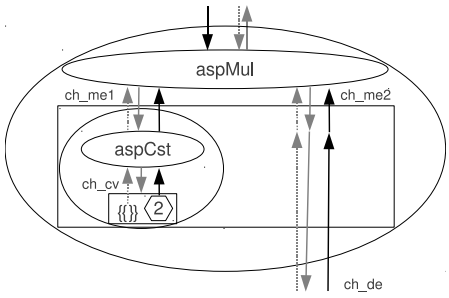

Fig. 10. aspDouble

Inside a macro a child can be mapped to some other macro ($rule_c$, *chMap*), where the subindex c stands for child. This is the case of the definitions of *aspPyth* and *aspDouble*, graphically represented in Figure 9 and Figure 10.

$$
\begin{aligned}
&lch_b \Longrightarrow (rule_c, chMap) = \lambda(sc_m, ic_m)\ (ic'0_m, ic0_b, sc0_b) \rightarrow \\
&\quad \textbf{let}\ (Fam\ ic'_c\ sp'_c) = agMacro\ (rule_c, chMap)\ (Fam\ sc_m\ (ic'0_m \mathbin{\#} lch_b)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (Fam\ ic_m\ \ emptyRecord) \\
&\qquad ic'1_m = hLeftUnion\ ic'_c\ (hDeleteAtLabel\ lch_b\ ic'0_m) \\
&\qquad ic1_b \;\; = lch_b \mathbin{.=.} emptyRecord \mathbin{.*.} ic0_b \\
&\qquad sc1_b \;\; = lch_b \mathbin{.=.} sp'_c \qquad\qquad\quad \mathbin{.*.} sc0_b \\
&\quad \textbf{in}\;\; (ic'0_m, ic1_b, sc1_b)
\end{aligned}
$$

In this case, the inner macro has to be evaluated using *agMacro*. The children of the inner macro will be included in the children of the outer macro; thus the synthesized attributes of the inner macro are included in sc_m, and the new inherited attributes of the children have to extend ic_m. The inherited attributes of the parent of the inner macro are the inherited attributes of the child *lch_b* of the base rule of the outer macro. The synthesized attributes of the parent of the inner macro are initialized with an empty attribution. The child lch_b is removed from $ic'0_m$, because the macro rule will not include it. On the other hand, the inherited attributes of the children of the inner macro (ic'_c) have to be added to the inherited attributes of the children of the macro. With the function *hLeftUnion* from *HList* we perform an union of records, choosing the elements of the left record in case of duplication. We initialize the inherited attributes for lch_b with an empty attribution, since it cannot be seen "from the outside". The synthesized attributes are initialized with the resulting synthesized attributes of the inner rule.

With the combinator ($\rightsquigarrow$) we define a mapping from a child with label *lch* to a literal value *cst*. For the base rule, the initial synthesized attributes of the child lch_b are fixed to the literal *cst*.

```
lch_b ⇝ cst = λ(_, _) (ic'0_m, ic0_b, sc0_b) →
  let ic'1_m = hDeleteAtLabel lch ic'0_m
      ic1_b  = lch_b .=. emptyRecord .*. ic0_b
      sc1_b  = lch_b .=. cst         .*. sc0_b
  in  (ic'1_m, ic1_b, sc1_b)
```

3.2 Attribute Redefinitions

We have now reached the point where we can use the described representations to show how we can extend and change the semantics associated with productions by joining in new language definitions. In the rest of this section we introduce some functions to *redefine* existing attributes.

In some cases we could want to introduce a specialized behavior to some specific attributes of an aspect defined by a macro. For example, the pretty printing attribute *spp* of the macros of Figure 6 currently is expressed in terms of the base rule. Thus when pretty printing *square* x, instead $x * x$ will be shown. Fortunately, given the nature of our approach, it turns out to be very easy to redefine specific attributes: we just update an existing attribute instead of adding a new one.

The function *synmod* (and its respective monadic version *synmodM*) modifies the definition of an existing synthesized attribute:

```
synmod att val (Fam ic sp) = Fam ic (hUpdateAtLabel att val sp)
```

Note that the only difference between *syndef*, from subsection 2.2, and *synmod*, is that the latter updates an existing field of the attribution *sp*, instead of adding a new field. With the use of the *HList*'s function *hUpdateAtLabel* we enforce (by type class constraints) the record *sp*, containing the synthesized attributes of the parent constructed thus far, to have a field labeled *att*. Thus, a rule created using *synmod* has to extend, using *ext*, some other rule that has already defined the synthesized attribute this rule is *redefining*. In Figure 11 we show how the pretty printing attributes of the language extensions we defined in Figure 6 can be redefined to reflect the input program:

```
sppSq  = synmodM spp $ do de ← at ch_de
                          return $ "square" >#< de # spp
aspSq' = sppSq `ext` aspSq
sppPyth  = synmodM spp $ do pe1 ← at ch_pe1
                            pe2 ← at ch_pe2
                            return $ "pyth" >#< pe1 # spp >#< pe2 # spp
aspPyth' = sppPyth `ext` aspPyth
sppDouble  = synmodM spp $ do de ← at ch_de
                              return $ "double" >#< de # spp
aspDouble' = sppDouble `ext` aspDouble
```

Fig. 11. Redefiniton of the *spp* attribute

The *AspectAG* library also provides functions *inhmodM* and *inhmod*, analogous to *inhdefM* and *inhdef*, that modify the definition of an inherited attribute for all children coming from a specified collection of semantic categories.

4 Conclusions and Future Work

We have introduced a set of combinators to define extensions to semantics expressed as first class attribute grammars. The programmer of the extensions does not need to know the details of the implementation of every attribute. In order to implement a macro or a redefinition for a production only the names of the children of the production are needed, which are provided in the definition of the abstract syntax tree, and the names of the attributes used.

This work is part of a bigger plan, involving the development of a series of techniques [1, 2, 11, 12] to deal with the problems involved in both syntactic and semantic extensions of a compiler in a type-safe way. We already think that the current approach is to be preferred over stacking more and more monads when defining a compositional semantics.

References

1. Arthur Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed abstract syntax. In *Fourth ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 15–26, New York, USA, 2009. ACM.
2. I. Baars, Arthur, Doaitse Swierstra, S., and Marcos Viera. Typed transformations of typed grammars: The left corner transform. In *Proceedings of the 9th Workshop on Language Descriptions Tools and Applications*, ENTCS, pages 18–33, 2009.
3. Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008.
4. Oege de Moor, Kevin Backhouse, and S. Doaitse Swierstra. First-class attribute grammars. *Informatica (Slovenia)*, 24(3), 2000.
5. Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 93–104, New York, NY, USA, 2009. ACM.
6. Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.
7. P. Jones, Mark. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.
8. Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.
9. B. M. Leavenworth. Syntax macros and extended translation. *Commun. ACM*, 9(11):790–793, 1966.
10. Conor McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, 2002.
11. S. Doaitse Swierstra. Parser combinators: from toys to tools. In Graham Hutton, editor, *Haskell Workshop*, 2000.
12. S. Doaitse Swierstra. Combinator parsing: A short tutorial. In *LerNet ALFA Summer School*, pages 252–300, 2008.
13. Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: how to do aspect oriented programming in haskell. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 245–256, New York, NY, USA, 2009. ACM.